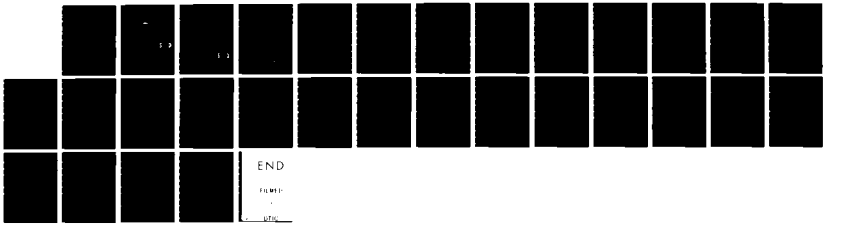
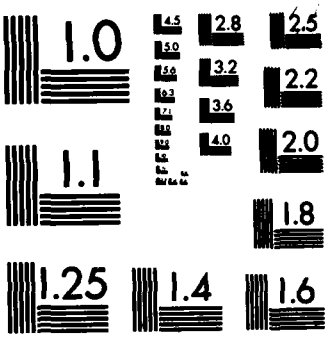
AD-A131 731 A MECHANICAL PROOF OF THE UNSOLVABILITY OF THE HALTING PROBLEM(U) TEXAS UNIV AT AUSTIN INST FOR COMPUTING SCIENCE AND COMPUTER A.. R S BOYER ET AL. JUL 82 1/1

UNCLASSIFIED ICSCA-CMP-28 N00014-81-K-0634 F/G 9/2 NL





SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: ICSCA-CMP-28	2. GOVT ACCESSION NO.: AD-A131731
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): A Mechanical Proof of the Unsolvability of the Halting Problem	5. TYPE OF REPORT & PERIOD COVERED: Technical
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Robert S. Boyer and J Strother Moore	8. CONTRACT OR GRANT NUMBER(s): MCS-8202943; N00014-81-K-0634
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Institute for Computing Science and Computer Applications / University [redacted] at Austin, Main Building 2100, Austin, Texas 78712	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: NR 049-500
11. CONTROLLING OFFICE NAME AND ADDRESS: Software Systems Science, National Science Foundation, Washington, D.C. 20550; Office of Naval Research, 800 N. Quincy Street, Arlington, VA 22217	12. REPORT DATE: July 1982 13. NUMBER OF PAGES: 26 Pages
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Reproduction in whole or in part is permitted for any purposes of the United States government.

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

DTIC ELECTE AUG 24 1983 S D B

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

automatic theorem-proving, interpreters, LISP, program verification, recursive unsolvability, termination

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The authors describe a proof by a computer program of the unsolvability of the halting problem. The halting problem is posed in a constructive, formal language. The computational paradigm formalized is Pure LISP, not Turing machines. We believe this is the first instance of a machine proving that a given problem is not solvable by machine.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Technical Report No. ICSCA-CMP-28
Grant No. MCS-8202943
Contract No. N00014-81-K-0634; NR 049-500

A MECHANICAL PROOF OF THE UNSOLVABILITY OF THE HALTING PROBLEM

Robert S. Boyer and J Strother Moore
Institute for Computing Science and Computer Applications
The University of Texas at Austin
Austin, TX 78712

July, 1982

Technical Report

Prepared for:

National Science Foundation
Software Systems Science Program
Washington, D.C. 20550

Office of Naval Research
800 N. Quincy Street
Arlington, VA 22217

Table of Contents

Abstract

We describe a proof by a computer program of the unsolvability of the halting problem. The halting problem is posed in a constructive, formal language. The computational paradigm formalized is Pure LISP, not Turing machines. We believe this is the first instance of a machine proving that a given problem is not solvable by machine.

1. Summary

Our current theorem-proving system, a descendant of systems described in [1] and [2], has proved that no computer program can decide whether a given program halts on a given input. To lead the theorem-prover to the proof, we suggested nine definitions and ten lemmas; our input to the theorem-prover is presented in section 6. To our knowledge, this is the first mechanical proof of the recursive unsolvability of any problem.

The model of computation used in our statement of the halting problem, described in section 2, is Pure LISP, not Turing machines. The unsolvability theorem is proved in a constructive logic like those of Skolem [5] and Goodstein [3], a logic that does not provide for bound variables ranging over infinite domains. The logic is briefly sketched in the appendix.

In section 3 we present a constructive statement of the unsolvability of the halting problem. Sections 4 and 5 contain an informal version of the proof.

The proof is an example of program verification via interpretive semantics.

We ask the reader, before he continues, to imagine a machine checkable proof of the unsolvability of the halting problem complete in every detail. For example, if the Turing machine approach is adopted, then, among many other details, one must contemplate the Godelization of Turing machines necessary to

pass one machine as an argument to another.

2. The LISP Interpreter

The programming language used in our statement of the halting problem is a version of Pure LISP [4]. We present our version by defining the logical function EVAL, which takes four arguments:

1. an S-expression to be evaluated,
2. a variable alist[1] assigning values to variable symbols,
3. a function alist assigning definitions to nonprimitive function symbols, and
4. a natural number, indicating the maximum depth of function calls.

EVAL returns either the value of the S-expression in the given environment or else it returns the object (BTM).

(BTM) is an object in the logic, axiomatized as an element of a "new" type using the shell principle. (BTM) is recognized by the function BTMP, which returns true or false according to whether its argument is (BTM). Furthermore, (BTM) is not equal to true, false, or any number, literal atom, or CONS. Thus (IF (BTM) 1 2) = 1. The reader is cautioned against thinking that a logical term involving (BTM) is necessarily (BTM).

We describe EVAL in the next three subsections. In the first we present EVAL formally. In the second (page 5) we paraphrase the formal definition in English. In the third (page 6) we give some example S-expressions and the values assigned by EVAL. These subsections may be read in any order.

---

[1] An alist is a list of pairs.

2.1. Formal Description of EVAL

Formally, EVAL is defined to satisfy the equation below. The formal logic used is sketched in the appendix. The functions GET, EVLIST, SUBRP, APPLY.SUBR, and PAIRLIST, used in the equation, are discussed informally below and defined formally in section 6.

```
(EVAL X VA FA N)
   =
(IF (NLISTP X)
    (IF (NUMBERP X)
        X
        (IF (EQUAL X 'T)
            T
            (IF (EQUAL X 'F)
                F
                (IF (EQUAL X NIL)
                    NIL
                    (GET X VA)))))
(IF (EQUAL (CAR X) 'QUOTE)
    (CADR X)
(IF (EQUAL (CAR X) 'IF)
    (IF (EQUAL (EVAL (CADR X) VA FA N)
               (BTM))
        (BTM)
        (IF (EVAL (CADR X) VA FA N)
            (EVAL (CADDR X) VA FA N)
            (EVAL (CADDDR X) VA FA N)))
(IF (EQUAL (EVLIST (CDR X) VA FA N)
           (BTM))
    (BTM)
(IF (SUBRP (CAR X))
    (APPLY.SUBR (CAR X)
                (EVLIST (CDR X) VA FA N))
(IF (EQUAL (GET (CAR X) FA)
           (BTM))
    (BTM)
(IF (ZEROP N)
    (BTM)
    (EVAL (CADR (GET (CAR X) FA))
          (PAIRLIST (CAR (GET (CAR X) FA))
                    (EVLIST (CDR X) VA FA N))
          FA
          (SUB1 N)))))))))).
```

The function GET takes two arguments. The second is understood to be an alist. GET looks up its first argument in the alist and returns the associated value if one is found. Otherwise, GET returns (BTM).

(EVLIST L VA FA N) treats L as a list of S-expressions, x_1, ..., x_k, and returns the list of their values

(LIST (EVAL x_1 VA FA N) ... (EVAL x_k VA FA N)).

However, should any x_i evaluate to (BTM), EVLIST returns (BTM).

Strictly speaking, our logic prohibits the definition of mutually recursive functions such as EVAL and EVLIST. The actual definitions of EVAL and EVLIST, which are presented in section 6, are preceded by the definition of a function EV which has five arguments, the first being used as a flag. Then (EVAL X VA FA N) is defined to be (EV 'AL X VA FA N) and (EVLIST X VA FA N) is defined to be (EV 'LIST X VA FA N). The admissibility of EV under the principle of definition follows from the observation that in each recursion either the last argument decreases or it stays even and the size of the second argument decreases.

(SUBRP X) returns true or false according to whether X is a member of '(ZERO TRUE FALSE ADD1 SUB1 NUMBERP CONS CAR CDR LISTP PACK UNPACK LITATOM EQUAL LIST). These are the primitives, other than 'IF and 'QUOTE, interpreted by EVAL.

APPLY.SUBR takes two arguments, the name of a primitive and a list of arguments, and returns the result of "applying" the primitive to the arguments. For example, (APPLY.SUBR 'CONS L) is (CONS (CAR L) (CADR L)) and (APPLY.SUBR 'LIST L) is L. For the purposes of the unsolvability proof obtained here it is necessary that CONS and LIST be among the primitives recognized by SUBRP and interpreted as above by APPLY.SUBR. Within these restrictions, arbitrary other names could be recognized by SUBRP and interpreted by APPLY.SUBR.

Finally, PAIRLIST takes two arguments. It pairs successive elements from the first with those from the second until the first list is empty. PAIRLIST

returns the list of such pairs. Thus, (PAIRLIST '(A B C) '(1 2 3)) is '((A . 1) (B . 2) (C . 3)).

2.2. An English Paraphrase of EVAL

To determine the value of an S-expression, X, under the variable alist VA, function alist FA, and maximum function call depth N, EVAL uses the following rules:

If X is not a list,
then
if X is a number, its value is X;
if X is the atom 'T, its value is true;
if X is the atom 'F, its value is false;
if X is the atom 'NIL, its value is 'NIL;
otherwise, X is treated as a variable symbol and
its value is found by looking it up in VA.

Otherwise, X is a list. Let fn be the first element of X and let x_1, ..., x_n be the remaining elements, which we will call the "actual expressions."

If fn is 'QUOTE, the value of X is x_1.

If fn is 'IF, the value of X is (BTM) if the value of x_1 is (BTM) and otherwise the value of X is either the value of x_3 or of x_2, according to whether the value of x_1 is false. Thus, our conditional is a 3-place IF that tests against false instead of an n-place COND that tests against NIL.

Otherwise, evaluate the actuals of X, x_1, ..., x_n, under the current VA, FA, and N. If any actual evaluates to (BTM), the value of X is (BTM).

If fn is a primitive function name, the value of X is obtained by applying the appropriate primitive function to the evaluated actuals.

Otherwise, look for a definition of fn on FA.
If no definition is found, the value of X is (BTM).
If a definition is found, it consists of two parts,
a list, called the formals of fn, and an s-expression,
called the body of fn.

If the maximum function call depth, N, is 0 (or not a natural number), the value of X is (BTM)

Otherwise, form a new variable alist by pairing the formals of fn with the evaluated actuals. The value of X is then the value of the body of fn under the new variable alist, the current function alist, FA, and maximum function call depth N-1.

2.3. Examples of EVAL

We now illustrate the programming language defined by EVAL. We do so by displaying some simple theorems about EVAL which show the values of various s-expressions in various environments.

Let v be the following variable alist, in which 'A has the value '(1 2 3) and 'B has the value '(A B C D):

```
v.   '((A . (1 2 3)) (B . (A B C D))).
```

Let w be the following, slightly different, variable alist, in which 'A has the value 0 and 'B has the value '(A B C D):

```
w.   '((A . 0) (B . (A B C D))).
```

Let f be the following function alist, defining the program APP:

```
f.   '((APP (X Y)
            (IF (EQUAL X NIL)
                Y
                (CONS (CAR X)
                      (APP (CDR X) Y))))).
```

Then the following equalities are theorems:

```
1.   (EVAL  5 v f N)                 = 5.

2.   (EVAL 'A v f N)                 = '(1 2 3).

3.   (EVAL '(QUOTE (E . 3)) v f N)  = '(E . 3).

4.   (EVAL '(IF A T F) v f N)        = T.

5.   (EVAL '(CONS 7 NIL) v f N)     = '(7).

6.   (EVAL '(IF X 1 2) v f N)        = (BTM).
```

```
7.  (EVAL '(APP A B) v f N)        = (IF (LESSP N 4)
                                         (BTM)
                                         '(1 2 3 A B C D)).

8.  (EVAL '(APP A B) w f N)        = (BTM).
```

A proof of Theorem 4 depends on the fact that '(1 2 3) is not F and that the value of the literal atom 'T is T. Theorem 6 may be proved from the observation that 'X is not given a value by the variable alist v. Theorem 7 informs us that under variable alist v and function alist f, '(APP A B) evaluates to (BTM) if the maximum function call depth is less than 4, and evaluates to '(1 2 3 A B C D) for all other depths. On the other hand, Theorem 8 informs us that under the variable alist w, '(APP A B) evaluates to (BTM) for all function call depths. A proof of Theorem 8 may be constructed from the observations that the value of 'A in w is 0, 0 is not NIL, and the CDR of 0 is 0.

3. The Halting Problem

Given an expression X it is not usually meaningful to ask whether it halts. One must consider whether it halts when evaluated under a particular variable alist and function alist.

When we say "the evaluation of X under VA and FA halts" we mean that there exists an n such that (EVAL X VA FA n) is not (BTM). Similarly, to say "the evaluation of X under VA and FA does not halt" means no such n exists, i.e., for all n (EVAL X VA FA n) is (BTM). We have seen that '(APP A B) under the variable alist

```
'((A . (1 2 3))
  (B . (A B C D)))
```

halts, while under the variable alist:

```
'((A . 0)
  (B . (A B C D)))
```

it does not halt.

To solve the halting problem, we desire a function alist containing a definition of a program named 'HALTS and its subroutines. 'HALTS must have the following properties. As input 'HALTS must take three arguments, an expression, x, and two alists, va and fa. Given a sufficient function call depth, the evaluation of a call of 'HALTS on such arguments must return either T or F. If the answer is T then the evaluation of x under va and fa should halt. If the answer is F then the evaluation of x under va and fa should not halt.

Let us now be more formal. Suppose we have in mind some function call depth N and some function alist FA on which 'HALTS is purportedly defined. Observe that

```
H.   (EVAL (LIST 'HALTS
                 (LIST 'QUOTE x)
                 (LIST 'QUOTE va)
                 (LIST 'QUOTE fa))
           NIL FA N)
```

is the value of 'HALTS when applied to some x, va, and fa (with function call depth N). If H is equal to F we will say that "'HALTS reports that x, va, and fa does not halt" and if H is equal to T we will say that "'HALTS reports that x, va, and fa does halt".

We want to prove that for every function alist FA and function call depth N, there exist x, va, and fa on which 'HALTS reports incorrectly. That is,

- if 'HALTS reports that x, va, and fa does not halt, i.e., H=F, then there exists a k such that (EVAL x va fa k) ≠ (BTM); and
- if 'HALTS reports that x, va, and fa halts, i.e., H=T, then for all K, (EVAL x va fa K) = (BTM).

Since ours is a constructive logic we must express this without the existential quantification over x, va, fa, and k. In particular, we must exhibit for any FA and N the required x, va, fa, and k. We therefore seek to express x, va, fa, and k as functions of FA and N. It suffices to define x,

va, and fa as functions of FA only and k as a function of N only. Given definitions of x, va, fa, and k, our statement of the unsolvability of the halting problem is:

```
HP. (IMPLIES
      (EQUAL H (EVAL (LIST 'HALTS
                           (LIST 'QUOTE (x FA))
                           (LIST 'QUOTE (va FA))
                           (LIST 'QUOTE (fa FA)))
                     NIL FA N))
      (AND
       (IMPLIES
           (EQUAL H F)
           (NOT (BTMP (EVAL (x FA) (va FA) (fa FA) (k N)))))
       (IMPLIES
           (EQUAL H T)
           (BTMP (EVAL (x FA) (va FA) (fa FA) K))))).
```

4. Definitions of x, va, fa, k

The functions x, va, fa, and k must be defined by the user of our theorem-prover before the unsolvability result can be posed to the theorem-prover. These definitions are the key to the unsolvability proof.

The intuitive idea behind the definition of x, va, and fa is: x should use 'HALTS to ask, of itself, "Does this program terminate?" and then either infinitely recur or not, in opposition to the answer supplied by 'HALTS. Therefore when x is evaluated under va and fa it must reconstruct x, va, and fa and call 'HALTS on those objects.

Let us attempt to meet these constraints by first considering the following list of two definitions:

```
'((CIRC (A)
      (IF (HALTS (QUOTE (CIRC A))
                 (LIST (CONS (QUOTE A)
                             A))
                 A)
          (LOOP)
          T))
  (LOOP NIL (LOOP))).
```

Let fa be defined to append this list to the front of FA, the function alist that purportedly solves the halting problem. Let x be defined to return the expression '(CIRC A), and let va return the singleton alist in which A is bound to fa (i.e., we pass as the argument to 'CIRC the definition of 'CIRC and its subroutines). The reader should confirm that if EVAL is applied to (x FA), (va FA), and (fa FA), the results of evaluating the arguments to 'HALTS inside 'CIRC are (x FA), (va FA), and (fa FA), as desired.

It remains to define k. If, with function call depth N, 'HALTS reports that (x FA) does not halt under (va FA) and (fa FA), we must exhibit a function call depth k sufficient for (x FA) to halt. Given our previous choices it is clear that k should be N+1.

Some readers could now "prove" HP. But HP is not a theorem, and a careful attempt to prove HP uncovers a technical flaw in our definitions. Consider what happens when 'HALTS is called inside 'CIRC. After the actuals are evaluated they are bound to the formals of 'HALTS and the resulting alist is used as the variable alist in the evaluation of the body of 'HALTS. But the function alist used is that containing 'CIRC, 'LOOP, and the definition of 'HALTS and its subroutines. How do we know that the evaluation of the body of 'HALTS will not be affected by the presence of our definitions for 'CIRC and 'LOOP? The answer is: we don't. Suppose the definition of 'HALTS on FA uses a subroutine named 'CIRC defined differently from above. Then our attempt to define 'CIRC will either overwrite the old definition of 'CIRC (causing 'HALTS to behave differently) or will be ignored (causing 'CIRC to behave differently) depending on whether we add our definition of 'CIRC to the front or the back of the function alist containing 'HALTS. A similar problem arises for 'LOOP.

However, here a lemma about EVAL can help us.

Lemma 1. Suppose that FN is a function name that does not occur as a

program name in the expression X and does not occur in the body of any function defined in a function alist FA. Let FA1 be FA with one additional definition on it, namely that of the function FN. Then (EVAL X VA FA1 N) is (EVAL X VA FA N).

Lemma 1 holds even if the result is (BTM). The proof is by induction on X and N. The actual version of this lemma proved in section 6 is a generalization concerning the function EV.

Thus, instead of choosing 'CIRC and 'LOOP as the names of our programs we should choose "new" names, names constructed from the given FA so as to be guaranteed not to occur in the body of 'HALTS or in any definition in FA. Since there is no requirement in our programming language that program names be atoms, it suffices to choose, in place of the name 'CIRC, the object (CONS FA 0), and, in place of 'LOOP, the object (CONS FA 1). It is straightforward to show that these names do not occur in FA.

Formal definitions of x, va, fa, and k are given in section 6. Note that 5 of the 10 lemmas in that section were stated to establish that the definitions of 'CIRC and 'LOOP do not interfere with the evaluation of 'HALT.

5. The Proof

We now prove HP. We use the following abbreviations:

```
x.        (x FA)
va.       (va FA)
fa.       (fa FA)
k.        (k N)
circ.     (CONS FA 0)
loop.     (CONS FA 1)
body.     (CADR (GET 'HALTS FA))
formals.  (CAR (GET 'HALTS FA)).
```

Recall that H is:

```
H.   (EVAL (LIST 'HALTS (LIST 'QUOTE x)
                        (LIST 'QUOTE va)
                        (LIST 'QUOTE fa))
           NIL FA N).
```

Observe that H is equal to:

```
H'.  (EVAL body
           (PAIRLIST formals
                     (LIST x va fa))
           FA
           (SUB1 N)),
```

unless N is 0 or 'HALTS is not defined on FA, in which case H is (BTM). Since we must consider only the two cases H=F and H=T, we conclude N is not 0, 'HALTS is defined on FA, and H is H'.

Case 1. H=F. We must show that (EVAL x va fa k) ≠ (BTM). By expanding the definition of EVAL and the code for circ

```
(EVAL x va fa k)
   =
(IF (BTMP h)
    (BTM)
    (IF h
        (EVAL (LIST loop) va fa N)
        (EVAL 'T va fa N))),
```

where h is:

```
h.   (EVAL body
           (PAIRLIST formals
                     (LIST x va fa))
           fa
           (SUB1 N)).
```

By two applications of Lemma 1 (one to remove the circ entry from fa and the next to remove the loop entry from fa) we get h=H'=H=F. Thus, (EVAL x va fa k) = (EVAL 'T va fa N) = T ≠ (BTM).

Case 2. H=T. We must show that (EVAL x va fa K) = (BTM). If K is less than 1, then (EVAL x va fa K) is (BTM). If K is 1 then the call of 'HALTS in the body of circ returns (BTM) so (EVAL x va fa K) is (BTM). Otherwise:

```
(EVAL x va fa K)
   =
(IF (BTMP h)
    (BTM)
    (IF h
        (EVAL (LIST loop) va fa (SUB1 K))
        (EVAL 'T va fa (SUB1 K)))),
```

where h is:

```
h.   (EVAL body
           (PAIRLIST formals
                     (LIST x va fa))
           fa
           (SUB1 (SUB1 K))).
```

By two applications of Lemma 1 we conclude that h=h':

```
h'.  (EVAL body
           (PAIRLIST formals
                     (LIST x va fa))
           FA
           (SUB1 (SUB1 K)))
```

Observe that in h' we have function call depth K-2 while in H' we have N-1. However, the following lemma establishes that h'=H' or else (BTMP h'):

Lemma 2. If both (EVAL X VA FA I) and (EVAL X VA FA J) are non-BTM they are equal. The proof is by simultaneous induction on X, I, and J.

Thus, h=h'=H'=H=T and hence (EVAL x va fa K) = (EVAL (LIST loop) va fa (SUB1 K)). However, Lemma 3, below, establishes that the latter EVAL is equal to (BTM).

Lemma 3. If fn is not a primitive function symbol and the body of fn on FA is (LIST fn), then (EVAL (LIST fn) VA FA N) is (BTM). The proof is by induction on N.

Thus, the unsolvability of the halting problem has been proved.

6. Input to the Theorem-Prover

We here present and annotate the commands typed to the theorem-prover which lead to the proof of the unsolvability of the halting problem. The theorem-prover responds to each theorem below with a proof and to each definition with a justification under the principle of definition.

The theorem-prover took 75 minutes of cpu time (running block compiled INTERLISP on a DEC 2060) to produce the proofs. Of this, 7 minutes were spent in garbage collection and 2 minutes were spent printing out the proofs.

6.1. The Definition of EVAL and Its Subroutines

1. Shell Definition.
Add the shell BTM of no arguments with
recognizer BTMP.

2. Definition.
```
(GET X ALIST)
   =
(IF (NLISTP ALIST)
    (BTM)
    (IF (EQUAL X (CAAR ALIST))
        (CDAR ALIST)
        (GET X (CDR ALIST))))
```

3. Definition.
```
(PAIRLIST X Y)
   =
(IF (NLISTP X)
    NIL
    (CONS (CONS (CAR X) (CAR Y))
          (PAIRLIST (CDR X) (CDR Y))))
```

4. Definition.
```
(SUBRP FN)
   =
(MEMBER FN
        '(ZERO TRUE FALSE ADD1 SUB1 NUMBERP CONS CAR
          CDR LISTP PACK UNPACK LITATOM EQUAL LIST))
```

5. Definition.

```
(APPLY.SUBR FN LST)
   =
(IF (EQUAL FN 'ZERO)        (ZERO)
(IF (EQUAL FN 'TRUE)        (TRUE)
(IF (EQUAL FN 'FALSE)       (FALSE)
(IF (EQUAL FN 'ADD1)        (ADD1 (CAR LST))
(IF (EQUAL FN 'SUB1)        (SUB1 (CAR LST))
(IF (EQUAL FN 'NUMBERP)     (NUMBERP (CAR LST))
(IF (EQUAL FN 'CONS)        (CONS (CAR LST) (CADR LST))
(IF (EQUAL FN 'LIST)        LST
(IF (EQUAL FN 'CAR)         (CAAR LST)
(IF (EQUAL FN 'CDR)         (CDAR LST)
(IF (EQUAL FN 'LISTP)       (LISTP (CAR LST))
(IF (EQUAL FN 'PACK)        (PACK (CAR LST))
(IF (EQUAL FN 'UNPACK)      (UNPACK (CAR LST))
(IF (EQUAL FN 'LITATOM)     (LITATOM (CAR LST))
(IF (EQUAL FN 'EQUAL)       (EQUAL (CAR LST) (CADR LST))
                            0))))))))))))))
```

6. Definition.

```
(EV FLG X VA FA N)
   =
(IF (EQUAL FLG 'AL)
    (IF (NLISTP X)
        (IF (NUMBERP X)    X
        (IF (EQUAL X 'T)   T
        (IF (EQUAL X 'F)   F
        (IF (EQUAL X NIL)  NIL
                           (GET X VA)))))
    (IF (EQUAL (CAR X) 'QUOTE)
        (CADR X)
    (IF (EQUAL (CAR X) 'IF)
        (IF (BTMP (EV 'AL (CADR X) VA FA N))
            (BTM)
            (IF (EV 'AL (CADR X) VA FA N)
                (EV 'AL (CADDR X) VA FA N)
                (EV 'AL (CADDDR X) VA FA N)))
    (IF (BTMP (EV 'LIST (CDR X) VA FA N))
        (BTM)
    (IF (SUBRP (CAR X))
        (APPLY.SUBR (CAR X)
                    (EV 'LIST (CDR X) VA FA N))
    (IF (BTMP (GET (CAR X) FA))
        (BTM)
    (IF (ZEROP N)
        (BTM)
        (EV 'AL
            (CADR (GET (CAR X) FA))
            (PAIRLIST (CAR (GET (CAR X) FA))
                      (EV 'LIST (CDR X) VA FA N))
            FA
            (SUB1 N)))))))))
(IF (LISTP X)
    (IF (BTMP (EV 'AL (CAR X) VA FA N))
        (BTM)
        (IF (BTMP (EV 'LIST (CDR X) VA FA N))
            (BTM)
            (CONS (EV 'AL (CAR X) VA FA N)
                  (EV 'LIST (CDR X) VA FA N))))
    NIL))
```

Hint. Consider the lexicographic order induced by LESSP and LESSP on (LIST N (COUNT X)).

7. Definition.

```
(EVAL X VA FA N)
   =
(EV 'AL X VA FA N)
```

8. Definition.
```
(EVLIST X VA FA N)
   =
(EV 'LIST X VA FA N)
```

6.2. The Definitions of x, va, fa, and k

We first define APPEND (so we can concatenate the definitions of 'CIRC and 'LOOP onto FA) and SUBLIS (so we can substitute new names for 'CIRC and 'LOOP).

9. Definition.
```
(APPEND X Y)
   =
(IF (NLISTP X)
    Y
    (CONS (CAR X) (APPEND (CDR X) Y)))
```

10. Definition.
```
(ASSOC VAR ALIST)
   =
(IF (NLISTP ALIST)
    F
    (IF (EQUAL VAR (CAAR ALIST))
        (CAR ALIST)
        (ASSOC VAR (CDR ALIST))))
```

11. Definition.
```
(SUBLIS ALIST X)
   =
(IF (NLISTP X)
    (IF (ASSOC X ALIST)
        (CDR (ASSOC X ALIST))
        X)
    (CONS (SUBLIS ALIST (CAR X))
          (SUBLIS ALIST (CDR X))))
```

12. Definition.
```
(x FA)
   =
(SUBLIS (LIST (CONS 'CIRC (CONS FA 0)))
        (QUOTE (CIRC A)))
```

```
13.  Definition.
     (fa FA)
        =
     (APPEND (SUBLIS (LIST (CONS 'CIRC (CONS FA 0))
                           (CONS 'LOOP (CONS FA 1)))
                     '((CIRC (A)
                        (IF (HALTS (QUOTE (CIRC A))
                                   (LIST (CONS (QUOTE A) A))
                                   A)
                            (LOOP)
                            T))
                       (LOOP NIL (LOOP))))
             FA)

14.  Definition.
     (va FA)
        =
     (LIST (CONS 'A (fa FA)))

15.  Definition.
     (k N)
        =
     (ADD1 N)
```

6.3. Lemma 1

```
16.  Definition.
     (OCCUR X Y)
        =
     (IF (EQUAL X Y)
         T
         (IF (NLISTP Y)
             F
             (OR (OCCUR X (CAR Y))
                 (OCCUR X (CDR Y))))))

17.  Definition.
     (OCCUR.IN.DEFNS X LST)
        =
     (IF (NLISTP LST)
         F
         (OR (OCCUR X (CADDR (CAR LST)))
             (OCCUR.IN.DEFNS X (CDR LST)))))

18.  Theorem.  OCCUR.OCCUR.IN.DEFNS:
     (IMPLIES (AND (NOT (OCCUR.IN.DEFNS FN FA))
                   (NOT (BTMP (GET X FA))))
              (NOT (OCCUR FN (CADR (GET X FA))))))
```

```
19.  Theorem.  LEMMA1:
     (IMPLIES (AND (NOT (OCCUR FN X))
                   (NOT (OCCUR.IN.DEFNS FN FA)))
              (EQUAL (EV FLG X VA
                         (CONS (CONS FN DEF) FA)
                         N)
                     (EV FLG X VA FA N)))
```

We state the straightforward lemmas establishing that our chosen replacements for 'CIRC and 'LOOP are indeed "new." We then have the system prove as, Corollary 1, that the evaluation of the body of 'HALTS under fa is the same as under FA. This is the sole use we make of Lemma 1, but if we do not have the system prove this Corollary and then forget Lemma 1 it wastes time trying to use Lemma 1 frequently.

```
20.  Theorem.  COUNT.OCCUR:
     (IMPLIES (LESSP (COUNT Y) (COUNT NAME))
              (NOT (OCCUR NAME Y)))

21.  Theorem.  COUNT.GET:
     (LESSP (COUNT (CADR (GET FN FA)))
            (ADD1 (COUNT FA)))

22.  Theorem.  COUNT.OCCUR.IN.DEFNS:
     (IMPLIES (LESSP (COUNT FA) (COUNT NAME))
              (NOT (OCCUR.IN.DEFNS NAME FA)))

23.  Theorem.  COROLLARY1:
     (EQUAL (EV 'AL
                (CADR (GET 'HALTS FA))
                VA
                (CONS (CONS (CONS FA 0) DEF0)
                      (CONS (LIST (CONS FA 1)
                                  NIL
                                  (LIST (CONS FA 1)))
                            FA))
                N)
            (EV 'AL
                (CADR (GET 'HALTS FA))
                VA FA N))


24.  Disable LEMMA1.
```

6.4. Lemma 2

```
25.  Theorem.  LEMMA2:
     (IMPLIES (AND (NOT (BTMP (EV FLG X VA FA N)))
                   (NOT (BTMP (EV FLG X VA FA K))))
              (EQUAL (EV FLG X VA FA N)
                     (EV FLG X VA FA K)))
```

Lemma 2 in its most general form is not useful to the theorem-prover as a rewrite rule. Consequently, we state Corollary 2 -- the only version of Lemma 2 we will subsequently need -- and tell the theorem-prover to prove it by using Lemma 2.

```
26.  Theorem.  COROLLARY2:
     (IMPLIES (EQUAL (EV FLG X VA FA N) T)
              (EV FLG X VA FA K))
     Hint:  Use LEMMA2.
```

6.5. Lemma 3

```
27.  Theorem.  LEMMA3:
     (IMPLIES (AND (LISTP X)
                   (LISTP (CAR X))
                   (NLISTP (CDR X))
                   (LISTP (GET (CAR X) FA))
                   (EQUAL (CAR (GET (CAR X) FA)) NIL)
                   (EQUAL (CADR (GET (CAR X) FA)) X))
              (BTMP (EV 'AL X VA FA N)))
```

6.6. A Lemma to Expand EVAL on CIRC

We state a lemma that can be regarded as a command to expand the definition of EVAL when it is applied to '(CIRC A). The system's heuristics for expanding recursive functions fail to see the merit of converting a question about the relatively simple expression '(CIRC A) to a question about the more complex body of 'CIRC.

28. Theorem. EXPAND.CIRC:

```
(IMPLIES
    (AND (NOT (BTMP VAL))
         (NOT (BTMP (GET (CONS FN 0) FA))))
    (EQUAL (EV 'AL
               (CONS (CONS FN 0) (QUOTE (A)))
               (LIST (CONS 'A VAL))
               FA J)
           (IF (ZEROP J)
               (BTM)
               (EV 'AL
                   (CADR (GET (CONS FN 0) FA))
                   (PAIRLIST (CAR (GET (CONS FN 0) FA))
                             (EV 'LIST
                                 (QUOTE (A))
                                 (LIST (CONS 'A VAL))
                                 FA J))
                   FA
                   (SUB1 J)))))
```

6.7. The Unsolvability of the Halting Problem

29. Theorem. UNSOLVABILITY.OF.THE.HALTING.PROBLEM:

```
(IMPLIES
 (EQUAL H (EVAL (LIST 'HALTS
                      (LIST 'QUOTE (x FA))
                      (LIST 'QUOTE (va FA))
                      (LIST 'QUOTE (fa FA)))
                NIL FA N))
 (AND
  (IMPLIES
      (EQUAL H F)
      (NOT (BTMP (EVAL (x FA) (va FA) (fa FA) (k N)))))
  (IMPLIES
      (EQUAL H T)
      (BTMP (EVAL (x FA) (va FA) (fa FA) K))))).
```

7. Chronology

Our first mechanical proof of the unsolvability of the halting problem was different from the one described here because we formalized the theorem in terms of "computation traces" instead of with EVAL. In addition, our approach to the quantification problem was different: we instructed the theorem-prover to assume, as an axiom, a formula that claimed that 'HALTS solves the halting problem for all programs and then we used the theorem-prover to prove that

T=F. The proof was obtained in March, 1982. It took us four days to guide, command, and cajole the theorem-prover to this first proof of the unsolvability result. Users less familiar with the system's heuristics might still be trying to get the proof through.

In May, 1982 we defined EVAL and stated the problem as seen here. However, where 'LOOP is called now in the definition of 'CIRC we originally called 'CIRC recursively on A. The proof that this recursion did not terminate was somewhat more complicated than the proof that 'LOOP does not terminate.

In June, 1982, after presenting the proof at the General Electric sponsored Whitney Symposium on Computer/Information Science and Technology, we saw the simplification that would result from introducing 'LOOP. In addition, we changed the theorem-prover for the first time in connection with this problem to simplify some of the proofs it was producing.

Our initial attempts to define the trouble-making program 'CIRC were incorrect. It is easy to intuit the idea of arranging for the evaluation of x under va and fa to ask whether x under va and fa halts. It is harder to find a definition that correctly manages to reconstruct its calling environment. In addition, the unsolvability problem involves several different levels of QUOTE -- a notoriously difficult construct. Several of our initial hand proofs were erroneous (although all were meant to be formalizations of the sketch presented here) and the errors were uncovered by our initial attempts at mechanical proof.

Appendix A. An Informal Sketch of the Formal Theory

We use the prefix syntax of Church to write down terms. For example, we write (PLUS X Y) where others might write PLUS(X,Y) or X+Y.

Our logic is a quantifier free, first-order logic obtained from the propositional calculus with equality and function symbols by adding (a) axioms

for certain basic function symbols, (b) a rule of inference permitting proof by induction on lexicographic combinations of well-founded relations, (c) a principle of definition permitting the introduction of total recursive functions, and (d) the "shell principle" permitting the introduction of axioms specifying "new" types of inductively defined objects.

The basic function symbols are TRUE, FALSE, IF, and EQUAL. The first two are function symbols of no arguments and return distinct constants which are abbreviated T and F respectively. IF is a function symbol of three arguments and is axiomatized so that (IF X Y Z) is Z if X is F and is Y otherwise. EQUAL is a function symbol of two arguments and is axiomatized so that (EQUAL X Y) is T if X is Y and is F otherwise.

Using the principle of definition we introduce the functions AND, OR, NOT, and IMPLIES in terms of IF. For example, (NOT P) = (IF P F T).

Using the shell principle we axiomatize several commonly used inductively constructed types. Among them are:

1. Natural numbers. A natural number is either the constant (ZERO) or is constructed from another natural number with the "constructor" function ADD1. The function NUMBERP "recognizes" natural numbers in the sense that (NUMBERP X) is axiomatized to be T or F according to whether X is a natural number. The function SUB1 is the "accessor" for ADD1 in the sense that if I is a natural number then (SUB1 (ADD1 I)) = I. SUB1 returns (ZERO) on (ZERO) and on non-NUMBERP objects.

2. Ordered pairs. An ordered pair is constructed from any two objects by the constructor function CONS. LISTP recognizes ordered pairs. CAR and CDR are the accessors for CONS: (CAR (CONS X Y)) = X and (CDR (CONS X Y)) = Y. CAR and CDR are axiomatized to return (ZERO) on non-LISTP objects.

3. Literal atoms. A literal atom is constructed from any object by the constructor PACK. LITATOM recognizes literal atoms. UNPACK is the accessor for PACK. UNPACK returns (ZERO) on non-LITATOMs.

Each shell class is disjoint from the others. For example, it is an axiom

that if X is a NUMBERP then X is not a LISTP or a LITATOM.

With the introduction of each shell class we obtain a well-founded relation permitting proof by induction and the definition of recursive functions under our principle of definition.

We define LESSP recursively so that if I and J are NUMBERPs (LESSP I J) is T or F according to whether I is strictly less than J.

The function COUNT assigns a numeric size to each object composed of NUMBERPs, LISTPs, and LITATOMs. The size of a composite object is larger than the sum of the sizes of its components. For example, (COUNT (CONS X Y)) = 1+(COUNT X)+(COUNT Y).

A precise description of our theory is given by the combination of Chapter III of [1] and Section 3 of [2].

We now briefly discuss our notational conventions.

0 is an abbreviation of (ZERO); the positive decimal integer n is an abbreviation of the nest of n ADD1's around a 0.

Nests of CARs and CDRs are abbreviated with function symbols of the form C...A...D...R, e.g., (CADAR X) is an abbreviation of (CAR (CDR (CAR X))).

We provide a convention for abbreviating some of our LITATOM constants. If wrd is a sequence of ASCII characters satisfying the syntactic rules for a symbol in our logic[2] and the ASCII codes for the successive characters in wrd are c_1, ..., c_n, then 'wrd is an abbreviation for

---

[2]Roughly speaking a symbol is a string of upper or lower case alphanumeric or sign characters beginning with an alphabetic or sign character other than +, -, or dot. However, see pg. 133 of [2] for the precise definition.

(PACK (CONS c_1 ... (CONS c_n 0)...)).

Thus, 'ABC is an abbreviation of (PACK (CONS 65 (CONS 66 (CONS 67 0)))). 'NIL is further abbreviated NIL.

(LIST x_1 x_2... x_n) is an abbreviation for (CONS x_1 (LIST x_2 ... x_n)). (LIST) is an abbreviation of NIL.

Finally, we provide a convention for abbreviating certain LISTP constants. For example, (LIST (CONS 'A 2) (CONS 'B 0)) may be abbreviated '((A . 2) (B . 0)). We so abbreviate any object constructed entirely by repeated CONSes from natural numbers and those LITATOMs admitting the abbreviation convention noted above. If x is such an object we abbreviate x by a single quote mark (') followed by the "pname" of x as defined below. If x is a NUMBERP abbreviated by n, its pname is n. If x is a LITATOM abbreviated by 'wrd, its pname is wrd. Otherwise, x is a LISTP. Let x_1, x_2, ..., x_n be the CARs of x and of its successive LISTP CDRs. Let fin be the nth CDR of x (i.e., the first non-LISTP in the CDR chain). If fin is NIL then the pname of x is an open parenthesis followed by the pname of x_1, a space (or arbitrary amount of white space), the pname of x_2, a space, ... the pname of x_n, and a close parenthesis. If fin is non-NIL then the pname of x is as it would be had fin been NIL except that immediately before the close parenthesis there should be inserted a space, a dot, a space, and the pname of fin.

REFERENCES

1. R.S. Boyer and J S. Moore. A Computational Logic. Academic Press, New York, 1979.

2. R.S. Boyer and J S. Moore. Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures. In The Correctness Problem in Computer Science, R.S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.

3. R. Goodstein. Recursive Number Theory. A Development of Recursive Arithmetic in a Logic Free Equation Calculus. North-Holland Publishing Co., Amsterdam, 1957.

4. J. McCarthy, et al.. LISP 1.5 Programmer's Manual. The MIT Press, Cambridge, Massachusetts, 1965.

5. T. Skolem. The Foundations of Elementary Arithmetic Established by Means of the Recursive Mode of Thought, without the Use of Apparent Variables Ranging over Infinite Domains. In From Frege to Goedel, J. van Heijenoort, Ed.,Harvard University Press, Cambridge, Massachusetts, 1967.

DISTRIBUTION LIST

Defense Documentation Center (12 copies)
Cameron Station
Alexandria, VA 22314

Naval Research Laboratory (6 copies)
Technical Information Division
Code 2627
Washington, D.C. 20375

Office of Naval Research (2 copies)
Information Systems Program (437)
Arlington, VA 22217

Office of Naval Research
Code 200
Arlington, VA 22217

Office of Naval Research
Code 455
Arlington, VA 22217

Office of Naval Research
Code 458
Arlington, VA 22217

Office of Naval Research
Eastern/Central Regional Office
Bldg 114, Section D
666 Summer Street
Boston, MA 02210

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, IL 60605

Office of Naval Research
Western Regional Office
1030 East Green Street
Pasadena, CA 91106

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
Code RD-1
Washington, D.C. 20380

Naval Ocean Systems Center
Advanced Software Technology Div.
Code 5200
San Diego, CA 92152

Mr. E. H. Gleissner
Naval Ship Research
& Development Center
Computation and Mathematics Dept.
Bethesda, MD 20084

Captain Grace M. Hopper (008)
Naval Data Automation Command
Washington Navy Yard
Building 166
Washington, D.C. 20374

END

FILMED

9-83

DTIC